The Report committee for Samer Gerges Chucri certifies that this is the
Approved version of the following report:

Image Compression using Locally Sensitive Hashing

APPROVED BY

SUPERVISING COMMITTEE:

Alexandros G. Dimakis, Supervisor

Sujay Sanghavi

Image Compression using Locally Sensitive Hashing

by

Samer Gerges Chucri, B.E.

REPORT

Presented to the Faculty of the Graduate School
of The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN ENGINEERING

The University of Texas at Austin
May 2013

Dedicated to my parents, Gerges and Marie-Louise.

Image Compression using Locally Sensitive Hashing

by

Samer Gerges Chucri, MSE
The University of Texas at Austin, 2013
SUPERVISOR: Alexandros G. Dimakis

The problem of archiving photos is becoming increasingly important as image databases are growing more popular, and larger in size. One could take the example of any social networking website, where users share hundreds of photos, resulting in billions of total images to be stored. Ideally, one would like to use minimal storage to archive these images, by making use of the redundancy that they share, while not sacrificing quality. We suggest a compression algorithm that aims at compressing across images, rather than compressing images individually. This is a very novel approach that has never been adopted before. This report presents the design of a new image database compression tool. In addition to that, we implement a complete system on C++, and show the significant gains that we achieve in some cases, where we compress 90% of the initial data. One of the main tools we use is Locally Sensitive Hashing (LSH), a relatively new technique mainly used for similarity search in high-dimensions.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

The problem of archiving photos is becoming increasingly important as image databases are growing more popular, and larger in size. One could take the example of any social networking website, where users share hundreds of photos, resulting in billions of total images to be stored. Ideally, one would like to use minimal storage to archive these images, by making use of the redundancy that they share, while not sacrificing quality.

The problem of compressing across images that we propose is very novel, and has not been tackled before. Image compression algorithms compress each image individually, and do not make use of the redundancy that some images might share. A simple example is archiving two images that are very similar, or even identical: If we were to apply image compression algorithms on each image individually, and then archive the files, we would store twice the amount of data. This is something that we try to avoid by making use of that similarity, to reduce the total amount of data stored.

Our objective is to design and implement a system which takes as input a list of images, applies efficient algorithms to compress across images, and outputs

compressed data that represents the initial database of images.

In this report, we will present the tools needed and used by our system, and state or prove some results of interest. We also describe the implementation of our system, and pin-point the design parameters. After describing all the components and algorithms employed, we will simulate our system and show the different results and trade-offs that we encounter. We run our algorithm on different databases and show the gains that we achieve depending on the scenario.

1.2 Plan

Our main goal is to leverage the redundancy present across images, and use that to reduce the total data stored. To do so, we adopt a very natural approach, which can be described concisely on a high level: Our system will cut every image into small square blocks, followed by processing done on these smaller blocks. Ideally, if two blocks are identical or similar, we would only keep one copy of the original block and then a pointer to that block. Our system will finally contain a library of blocks, and a list of pointers. In summary this is the high level description of our system:

- System Data: A library Q of blocks, list P of pointers.

  For every image I in our database:

1. Cut image I into blocks of specific size

   For every block b in I

2. Apply a transformation ϕ on b: $\phi(b)$

3. Look for closest neighbor $\phi(b)$ in Q, called c

4. If c is 'close enough' to $\phi(b)$, then represent b by a pointer to c. Otherwise, add c to the library Q

The solution that we adopt is fairly simple, but we will show that it takes care of our objective of targeting the redundancy. The main part of our solution relies on efficiently looking for the closest neighbor of each block, which we will tackle in chapter 2. We will focus on locally sensitive hash functions, which will help us in solving that specific problem.

Chapter 2

Nearest-neighbor search

As outlined in chapter 1, nearest-neighbor search is one of the main components of our algorithm. We will present in this section several methods that can be used to perform this task and highlight their advantages and disadvantages.

2.1 Problem definition

The problem of finding the exact nearest neighbor can be described as follows: Given a list of N points $X = x_1, ..., x_N$ in d-dimensions and a query point q, return $x^* = argmin||x_i - q||$.

2.2 Exact nearest-neighbor search

2.2.1 Exhaustive search

The naive solution would be to scan all N points and compute the distance between every x_i and q. This solution is $O(N)$ and does not make use of the structure of these points. We can obviously do better than this.

2.2.2 K-d Trees

Kd-trees were introduced by Bentley in [1], and aim to make use of the spatial structure of the points in X. Informally, a Kd-tree partitions the dataset by carefully selecting hyperplanes and splitting the data into what is above and under the hyperplane. In the tree representation, every node represents a k-dimensional point with an associated hyperplane that passes through it, and the left subtree consists of points below the hyperplane, whereas the right subtree consists of points above the hyperplane. By choosing the hyperplane directions appropriately, one can achieve fast range queries, with average look up time $O(logN)$, and worst case $O(N)$, assuming a low dimensionality. In addition to the bad worst case complexity, K-d trees fail for high dimensional data: The actual running time can be shown to be $O(min(dN, e^d))$. This is obviously very bad for us as we will be dealing with dimensions larger or equal to 64.

2.3 Approximate nearest-neighbor search

In order to achieve better running time, one tries to approximate the original problem, by defining a new one which is more relaxed. This approach is very popular for NP-hard problems, where one comes up with an approximation algorithm that provides some probabilistic guarantees on the solution that it generates, while running in polynomial time. In our case, we relax the exact nearest neighbor search and define an approximate nearest neighbor. The approximate nearest neighbor problem is defined as follows:

Definition 2.3.1. An ϵ-approximate nearest neighbor (ϵ-ANN) of x, is any point y such that: $||y - x|| \leq (1 + \epsilon)||x - z|| \forall z \neq x$

An algorithm A that solves this problem, takes as input a list of N points $X = x_1, ..., x_N$ in d-dimensions and a query point q, and returns a point y in X such that y is an ϵ-ANN of q. Gionis and Indyk discuss the motivation of such a choice, where an approximate nearest neighbor answer is good enough [2]. The main tool to solve this problem efficiently are locally sensitive hash (LSH) functions. We will further relax the problem definition, by assuming that an algorithm A solves the ϵ-approximate nearest neighbor problem, if it returns an ϵ-ANN with high probability. Under these assumptions, it can be shown that the ϵ-ANN problem can be solved in sub-linear time $O(N^p), p < 1$, using LSH function families.

2.3.1 Locally Sensitive Hashing *(LSH)*

Locally sensitive hash functions first emerged in the context of similarity search in high-dimensions [3], [2]. Informally, locally sensitive hash (LSH) functions are hash functions that satisfy the following property: If x and y are 'close', then they hash to the same value with high probability, and if they are 'far' then they hash to different values with high probability. We define the concept of locally sensitive hashing more formally below.

Definition 2.3.2. A family H is called (r_1, r_2, p_1, p_2)-sensitive for $||.||$, if for any x, y:

- $Pr_H[h(x) = h(y)] \geq p_1$, if $||x - y|| \leq r_1$
- $Pr_H[h(x) = h(y)] \leq p_2$, if $||x - y|| \geq r_2$

We will refer to a family satisfying definition 2.3.2 as locally sensitive hash (LSH) function family. For this definition to be useful, we require $p_1 \geq p_2$ and $r_1 \leq r_2$. An ideal LSH function family would have p_1 as close to 1 as possible, and p_2 as close to 0 as possible. It is shown in [3] and [2], that a LSH function family can be used to solve the ϵ-ANN problem previously defined in sub-linear time. The main idea is to use a concept often referred to as amplification, with parameters K and L. Amplification results in a final hash function family G, with amplified gap between the probabilities p_1 and p_2, in the hope of approaching the ideal LSH function family. Note that alternate definitions of LSH exist [4], but are very closely related and similar.

2.3.1.1 Amplification

Given a LSH function family H, we form a LSH function family G as follows:

1. let $g_i(x) = [h_{i,1}(x)|h_{i,2}(x)|...|h_{i,K}(x)], 1 \leq i \leq L$, where $h_{i,j}$ is sampled uniformly at random from H.
2. Define equality as follows: $g(x) = g(y)$ iff $\exists i \in 1, ..., L$ such that $g_i(x) = g_i(y)$.

It is easy to prove that this procedure forms a LSH function family G that is $(r_1, r_2, 1-(1-p_1^K)^L, 1-(1-p_2^K)^L)$-sensitive. Amplification is a very essential process in the ANN search. Referring to every $g_i(x)$ as a bucket, we can say that every point x hashes to L buckets, namely $\{g_1(x), ..., g_L(x)\}$. Two points x and y hash to the same value if they hash to any common bucket. After appropriate amplification, two sufficiently close points will share at least one common bucket with high probability, and two sufficiently far points will not share any common bucket with high probability. Figure 2.1 shows the effect of amplification parameters on the probability of collision. Referring to the amplification curve as $f(p)$, our probabilities p_1 and p_2 become $f(p_1)$ and $f(p_2)$ respectively post to amplification. Thus, one can pick the parameters K and L to simultaneously push $f(p_1)$ to 1 and $f(p_2)$ to 0 as desired.

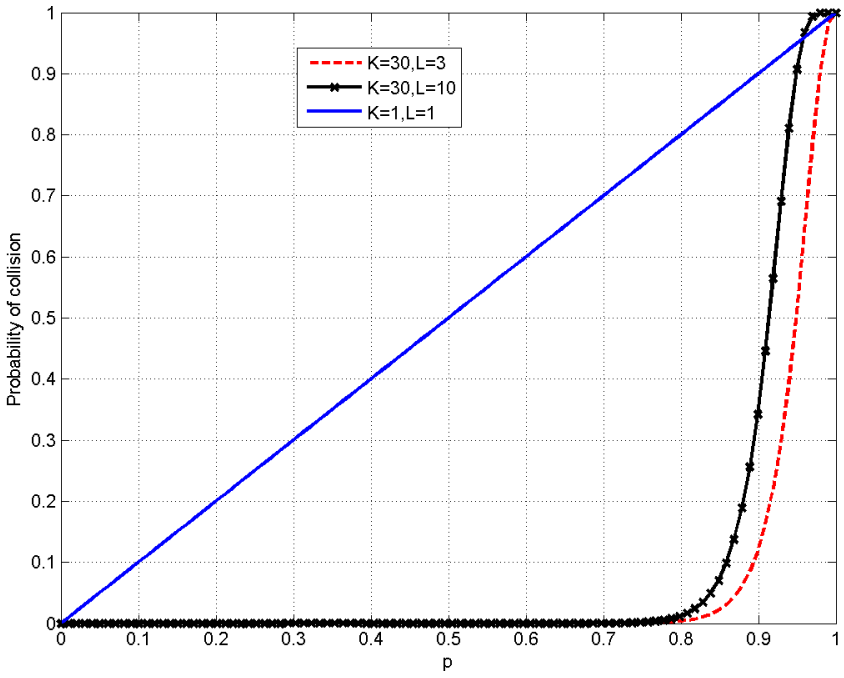

Figure 2.1: Amplification effect

Chapter 3

System Design

3.1 Overview

We built our program purely on C++, as we are targeting the fastest running time possible. Recall that after processing the images, we will be storing the following: A library Q of reference blocks, and a list P of pointers. During the processing step, we use different data structures, which will finally result in the desired compressed output Q and P. Before we describe the system components, it is helpful to summarize the steps that our program performs, and introduce the components along the way.

3.1.1 Algorithm steps

1. We start by scanning the jpeg files in the directory. After collecting a list of the files, we start processing each file individually, one after the other.

2. We decompress the jpeg image to obtain a raw image representation of the pixels, using libjpeg library.

3. We cut the image into blocks of 8×8 pixels, and represent them as arrays.

4. We compute the DCT of every block. We will discuss the choice of DCT over raw pixels in section 3.4.

5. For every DCT block, we compute the hash value v, and check if any of the other blocks in the library hash to the same value. This step is made very efficient, and will be discussed in more detail.

6. If any of the blocks that hash to v are close enough, we pick the closest block and point to it. Otherwise, we add our block to the library of reference blocks Q.

An important aspect in step 5, is how to efficiently search for the blocks in the library that hash to some particular value v, i.e. finding the collisions.

3.2 Finding Collisions

The problem can be formulated as follows: Given a hash value v, find the blocks in library that hash to v, if any exist.

3.2.1 Solution 1

One way is to loop through all the blocks in the library Q, compute their hash, and check if they are equal to v. This would take require us $|Q|t$ computations, where $|Q|$ is the cardinality of our library, and t the number of computations needed to evaluate the hash function. Obviously this is not a very attractive method although it requires no auxiliary memory.

3.2.2 Solution 2

A smarter and faster way to solve this problem is by using standard hash tables (hash maps). A hash table is a data structure that stores elements consisting of a pair *(key,value)*. A hash table uses standard hash functions (not locally sensitive hash functions) to provide efficient look up time, with expected $O(1)$ look up time. The *key* field is the id of the element, and the *value* field represents the data. In our case, the key is simply the hash value v (obtained from the LSH functions), and the *value* will consist of a list. This list contains the IDs of the blocks that hash to v. Given that our expected look up time is $O(1)$, the expected running time of this approach is proportional the the number of blocks in the library that hash to v. The only disadvantage is the memory storage that this method needs.

To apply solution 2, we modify step 6 in section 3.1.1 to the following: If any of the blocks that hash to v are close enough, we pick the closest block and point to it. Otherwise, we add our block to the library of reference blocks Q and to the hash table, by retrieving the element with key equal v if one exists, and appending our block to its list. If no such element exists, we create a new entry in the hash table.

3.3 LSH function

As explained earlier, a very important design parameter is the hash function family used. In this section, we will present a simple LSH function family that is sensitive to the L_1 norm. After proving this, we present the problems we might face

using this function, and the variations that we consider to come up with a modified LSH function family that serves our purpose.

3.3.1 L1 sensitive LSH function

We are mainly interested in the L_1 norm of our blocks, so we would like to pick a LSH family that is sensitive to this norm. We will introduce a simple LSH function family first and prove that it is indeed sensitive to that norm. Later, we show how to modify this to meet more sophisticated requirements that we set.

We assume that the components of the d-dimensional vector x lie in a set of finite cardinality. Our input is consistent with this assumption since our raw blocks represent pixels of different integer intensities. The intensities can range from 0 to 255. Also, the DCT coefficients are rounded and can be shown to have a fixed range between -1024 and 1024.

We consider the following function defined as:

$$h_{\alpha,\beta}(x) = \begin{cases} 1, & \text{if } x_\alpha < \beta \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

Notice that the function is uniquely defined by α and β.

The corresponding hash function family is:

$$H = \{h_{\alpha,\beta} | \alpha \in D, \beta \in R\} \tag{3.2}$$

Where $D = \{1, 2, ..., d\}$ and R is some set of finite cardinality. One example is $R = \{-1024, -1023, ..., 1024\}$.

Theorem 1. *The hash function family defined in (3.2) is* $(r_1, r_2, 1 - \frac{r_1}{d|R|}, 1 - \frac{r_2}{d|R|})$*-sensitive to the* L_1 *norm.*

Proof. In order to prove this, it is helpful to consider the following expression: $Pr_H(h(x) \neq h(y))$. Note that α and β are picked independently and uniformly at random from their range.

$$
\begin{aligned}
Pr_H(h(x) \neq h(y)) &= \sum_{\alpha=1}^{d} \sum_{\beta \in R} Pr(\alpha) Pr(\beta) Pr(h_{\alpha,\beta}(x) = h_{\alpha,\beta}(y)) && (3.3) \\
&= \sum_{\alpha=1}^{d} Pr(\alpha) \sum_{\beta \in R} Pr(\beta) Pr(min(x_\alpha, y_\alpha) \leq \beta \leq max(x_\alpha, y_\alpha) \\
&= \sum_{\alpha=1}^{d} \frac{1}{d} \frac{1}{|R|} |x_\alpha - y_\alpha| && (3.4) \\
&= \frac{1}{d} \frac{1}{|R|} \sum_{\alpha=1}^{d} |x_\alpha - y_\alpha| && (3.5) \\
&= \frac{1}{d|R|} ||x - y||_1 && (3.6)
\end{aligned}
$$

□

What is also appealing about this hash function is its simplicity to form and compute. The output is simply a bit value, and for a fixed choice of α and β, it takes only 1 comparison operation to determine its value. Thus it will be very easy to store and compute.

3.3.2 Problems faced

Although this function is simple and provides us with L_1 norm sensitivity, we will see in future sections that it would be smarter to modify the function by picking distributions for α and β that result in better running time.

3.4 Input Domain

After partitioning our picture into blocks, a natural question arises: should we compare the blocks in the time domain or DCT domain? We answer this question by analyzing the differences in the performance of these two methods, and the properties that each domain preserves. We establish that the DCT domain provides a much more compact description of our blocks, and support this by simulations. We considered a database of $1,502$ JPEG images, resulting in $179,151,36$ 64-dimensional blocks.

3.4.1 Distribution

We consider a blocksize of 8×8 pixels, resulting in 64 dimensional points. We study the distribution per component, in the raw pixel (time) domain and DCT domain.

3.4.1.1 Raw pixel domain

It is intuitive to think that every pixel gives us very little information about the whole block, and this has been established in theory. In natural images, pixels that are adjacent are expected to be very close and very much correlated, due to

the simple concept of continuity. To study the statistics of every pixel location out of the 64 possible ones, we cut every image into 8×8 blocks, and scan all the blocks, while updating the corresponding pixel statistics. We notice that all the pixel locations have a similar distribution: In figure 3.1, we show the distribution for 4 different pixel locations. Note that the range is quite large, covering all the region from 0 to 255.

3.4.1.2 DCT domain

It has been established that DCT provides an excellent basis for images for several reasons. In this section, we are concerned with the statistics of these components. The first thing that we notice in the DCT case is that the distribution for different components is not the same. Figure 3.2 shows the distribution of the first four coefficients *(Low frequency)*, whereas figure 3.3 shows the distribution of the last four coefficients *(High Frequency)*. Notice that the DC component behaves differently than other frequency components, and this is expected since it represents the average pixel value. Note that the shape here is very similar to the raw pixel domain case. For higher frequency components, we notice that all share the same distribution shape, however the variance shrinks as we shift into higher frequency components. This is also expected since in the quantization step ‚JPEG renders the higher frequency components very small by scaling their value, and then rounding. This observation is key, as it tells us that the description of every block in the DCT domain is compact and sparse.

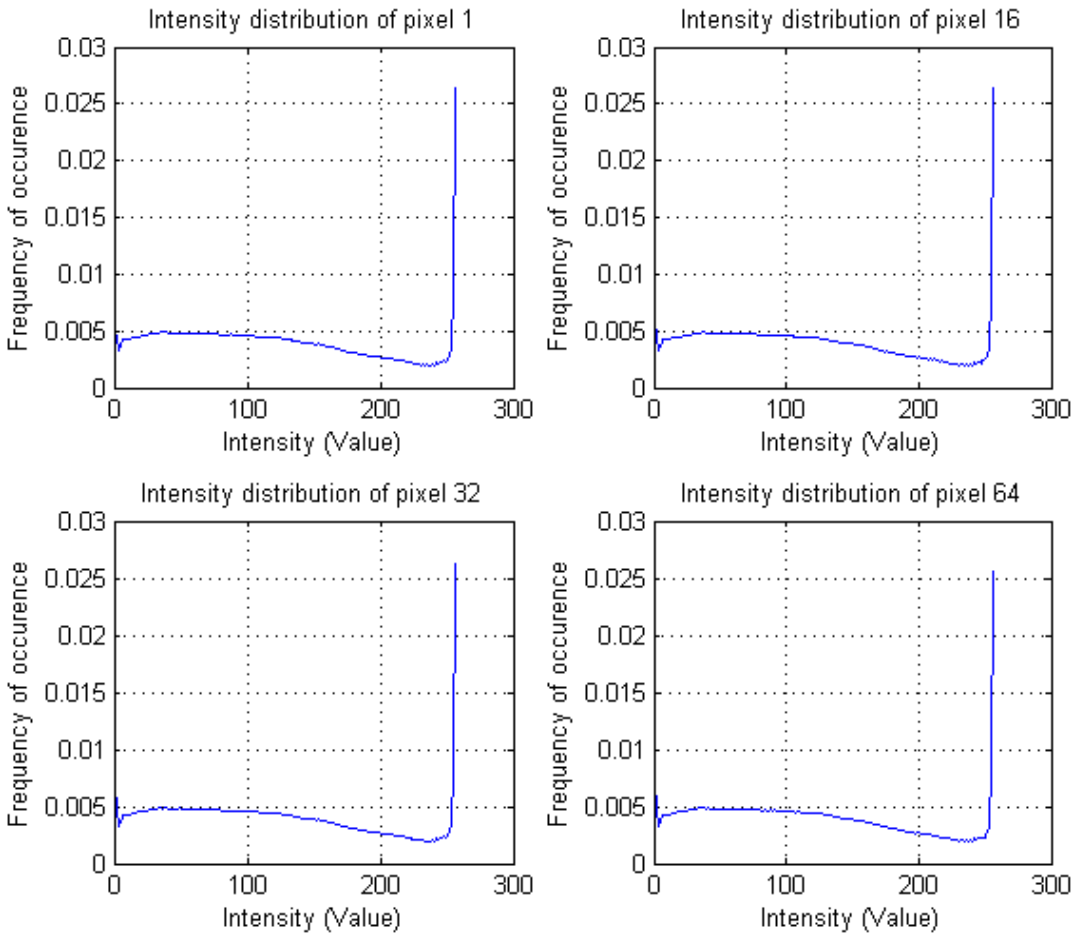

Figure 3.1: Intensity Distribution of different pixels

3.5 LSH function family revisited

Since we will be applying our LSH function to the DCT blocks, we suggest a new locally sensitive hash function family with adjusted parameters. DCT coefficients are not equally important: lower frequency components carry more weight in terms of visual content. Since we will be dealing with JPEG images, we know that

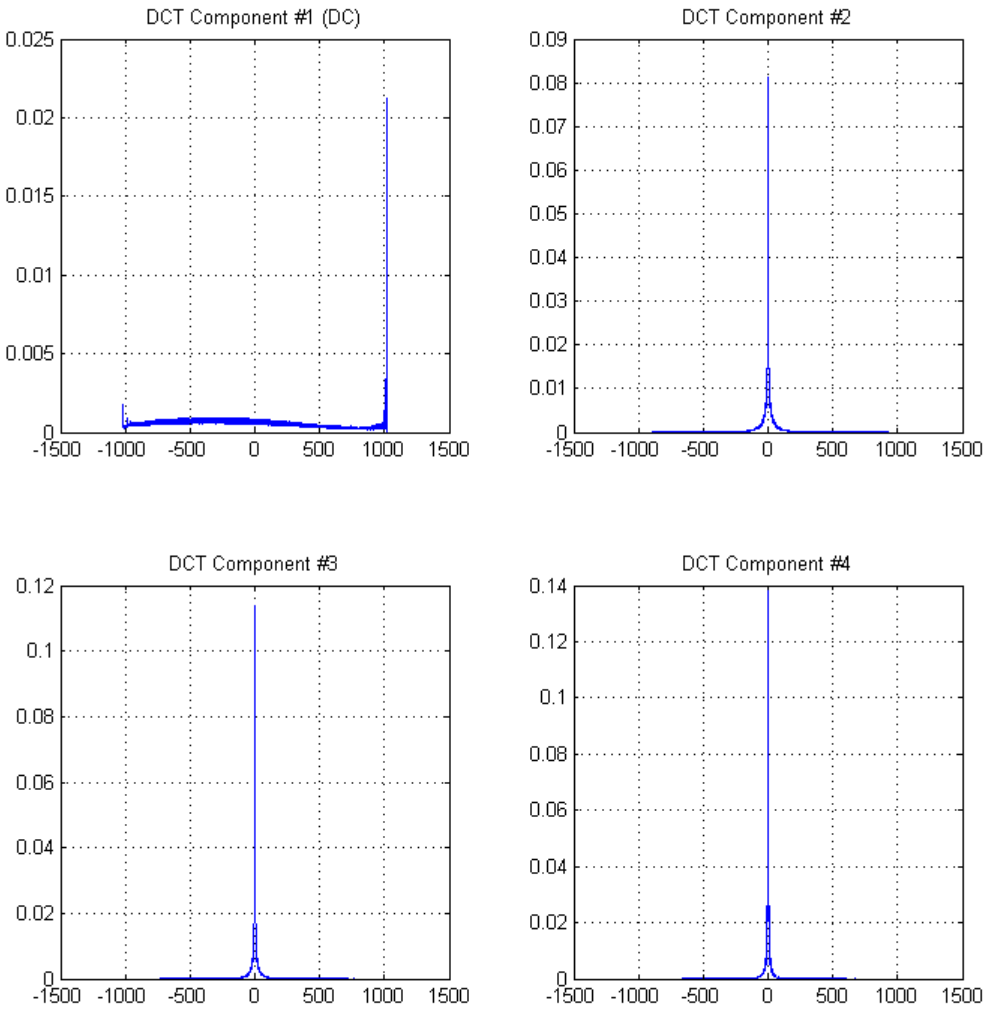

Figure 3.2: Distribution of first 4 DCT coefficients *(Low Frequency)*

most of the non-zero components will be the low frequency ones, which suggests a natural modification to our LSH function introduced in section 3.3. Instead of picking α uniformly at random, we pick it according to the quantization weights that

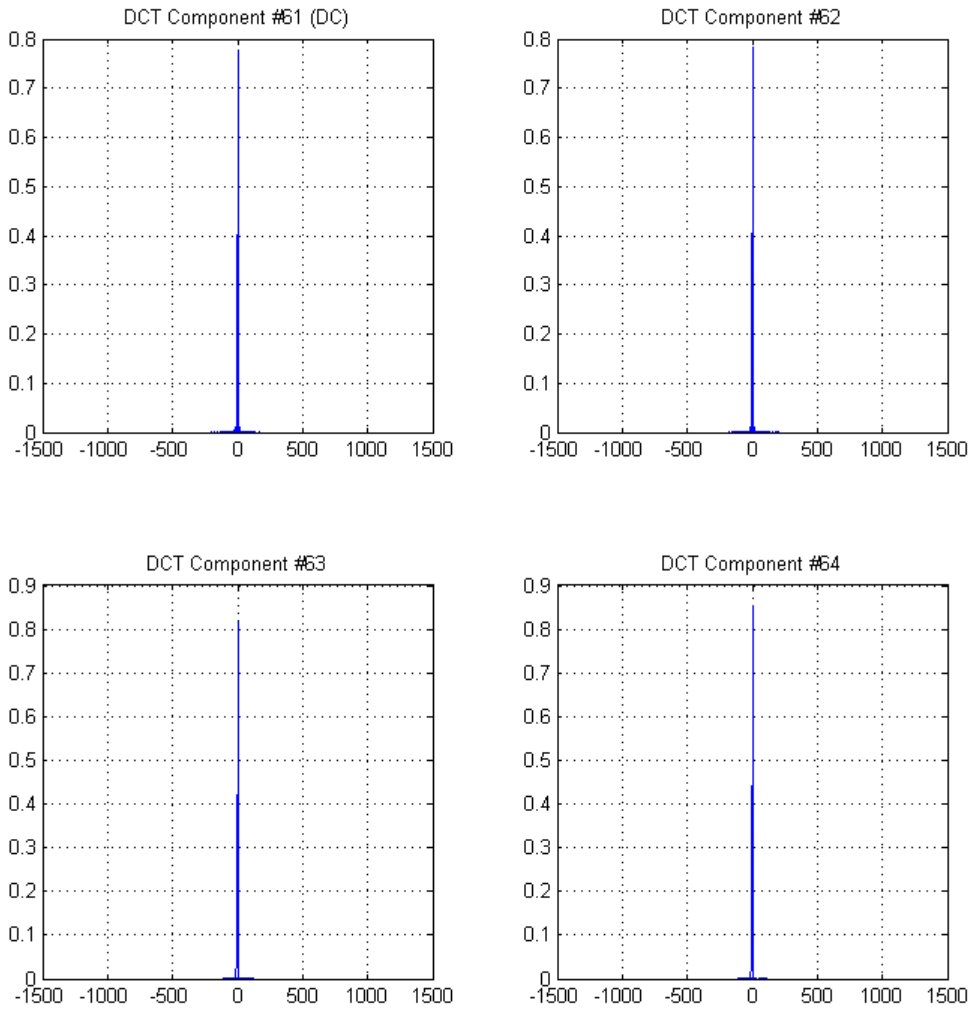

Figure 3.3: Distribution of last 4 DCT coefficients *(High Frequency)*

JPEG uses, which represent a measure of importance of the corresponding coeffi-

cient. This is often referred to as the quantization matrix M, and is shown below:

$$\begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix} \tag{3.7}$$

The lower the weight, the more significant a coefficient is, thus we define the weights as:

$$w_i = \frac{\frac{1}{M_i^2}}{\sum \frac{1}{M_j^2}} \tag{3.8}$$

where i is the i-th entry of the total 64 coefficients of M.

3.5.1 Choice of α

We pick α according to the weights w_i, such that α is equal to i with probability w_i. This changes the probability of collision of two blocks, which can be seen by inspecting $Pr_H(h(x) \neq h(y))$.

$$Pr_H(h(x) \neq h(y)) \quad = \quad \frac{1}{|R|} \sum_{\alpha=1}^{d} w_\alpha |x_\alpha - y_\alpha| \tag{3.9}$$

Now the LSH family is sensitive to the weighted L_1 norm of the blocks.

3.5.2 Choice of β

Since the DCT coefficients have limited range for higher frequency components, we can pick β over this desired range. Note that picking β outside this

range would not provide any information since all the blocks would hash to the same value, and this would ruin the running time of our algorithm. This point is studied in detail in chapter 4, where we show the effect of the distribution of β on the running time and quality of our compression algorithm.

Chapter 4

Results

In this section, we will run our system on different image databases and assess its performance while highlighting the different trade-offs observed. We have a lot of parameters that we can vary, and we will list these parameters in section 4.2. We consider running our algorithm on several types of databases, such as an artificial database that we construct, and a regular image database. It is worth noting that at this stage, we are running our algorithm on grayscale images for simplicity.

4.1 Performance criteria

Our system has a lot of variables that determine its performance. In this section we will list the main parameters involved that can affect the overall performance. Performance includes both running time and compression ratio.

4.1.1 Running Time

To quantify the running time, we will consider a very specific metric of our algorithm. The main component as discussed previously is the neighbor search: For every block b, we look for the nearest in the reduced space of blocks with same hash

value as b. The cardinality of that reduced space will be the number of comparisons that we need to make. For this purpose, the average number of comparisons per block is a good metric to determine the overall running time. In our plots, we will provide the average comparisons per block for every image added to our system.

4.1.2 Compression Ratio

In order to define the compression ratio, it is useful to recall what our system eventually stores: A library Q of reference blocks, and a list P of pointers. The size of pointers P can be assumed to be negligible compared the size of Q and thus we will mainly focus on the size of Q. As we are processing the images, we count the total number of blocks processed, which actually constitute all the images. We denote by T the total number of blocks inspected, and defined the compression ratio as follows:

$$C = \frac{|Q|}{|T|} \tag{4.1}$$

4.2 Parameters

In this section we will specify the main parameters of interest.

4.2.1 Hash function parameters

4.2.1.1 Amplification parameters

As mentioned previously, the amplification step requires two parameters: K and L. In our program, we have the choice of varying K between 1 and 32. The higher K is, the more we are restricting the probability of collision of blocks further

away from each other. On the other hand, L has the opposite role, where increasing L increases the probability of collision, but of course in a different manner. Looking at the amplification curve, picking K and L can be used to theoretically design the probability of collision as desired.

4.2.1.2 α and β distribution

As discussed earlier, we pick α with probability proportional to the importance of the corresponding DCT component. We will fix this distribution of α. On the other hand, β presents a very obvious trade-off between running time and bucket structure. Conditioned on the choice of α, a natural way is to pick β according to the distribution of that coefficient that we obtained via simulations, uniform around its mean, with range proportional to its standard deviation. A second choice is to simply pick β to be fixed, and equal to the mean or median of the distribution of that specific coefficient α. However, if we would like to ensure the property that buckets contain 'close' blocks, going with the first option is obviously better. This is further illustrated in the simulations.

4.2.2 Threshold θ

This parameter determines how lossy we allow our compression algorithm to be. In our algorithm, while processing block b, if the closest block found in our library is within our defined threshold, we reference b with a pointer. Increasing θ decreases the compression ratio *(more compression)*, but also decreases the quality of the images. Note that we define θ as a multiplicative factor times the dimension,

resulting in an actual threshold of 64θ.

4.3 Artificial Database Simulations Results

This experiment is just a toy example to motivate our algorithm. The main purpose is to show how our algorithm performs on a database of very similar images.

4.3.1 Generation

Given a grayscale image, we generate 100 versions of the image, where each is compressed using jpeg with a different quality factor. This means that the first picture will be compressed with quality 1, the second with quality 2,..., and the hundredth with quality 100. We run simulations for this database under two scenarios: In the first one, we fix β to the mean value, whereas in the second scenario, we let β be uniform over the standard deviation range.

4.3.2 Scenario I

4.3.2.1 Parameters

For the LSH amplification parameters we set K equal to 30 and L equal to 3. We let α be distributed according to the DCT coefficient weights, and β picked to be the mean value for the corresponding α coefficient. We also set the threshold θ equal to 2.

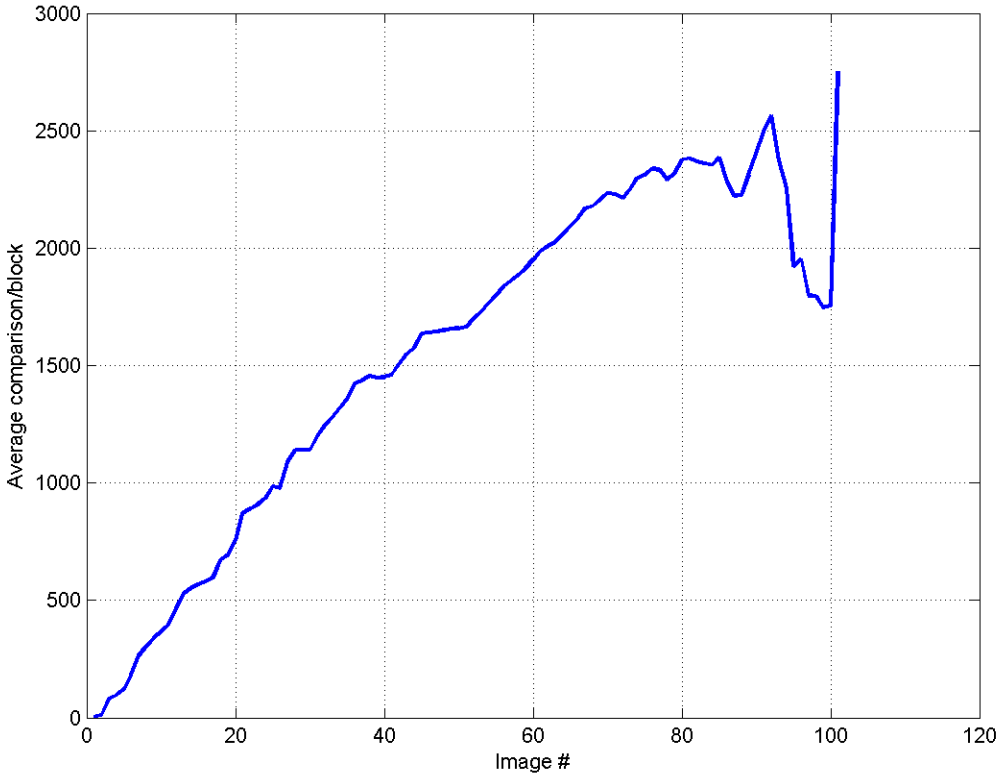

Figure 4.1: Average Comparisons per block, Scenario I

4.3.2.2 Running Time

The trend is clear: As the number of images increase, the number of blocks in library increase causing the average comparisons per block to increase. This can be clearly seen in figure 4.1.

4.3.2.3 Compression

After processing an image, some of its blocks are added to our library, and this is shown in figure 4.2. It is interesting to note that the slope of this curve decreases with the number of images added, since our library grows richer. As we keep on adding images, the compression ratio decreases as shown in figure 4.3, to finally reach a value around 0.16.

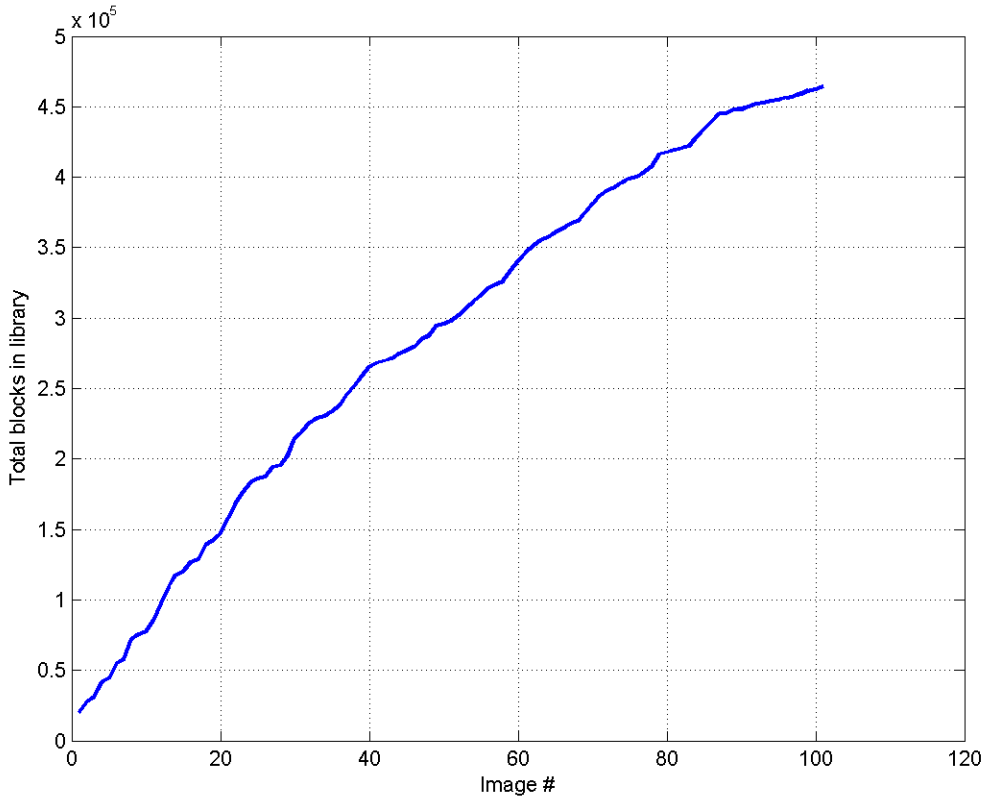

Figure 4.2: Number of blocks in library, Scenario I

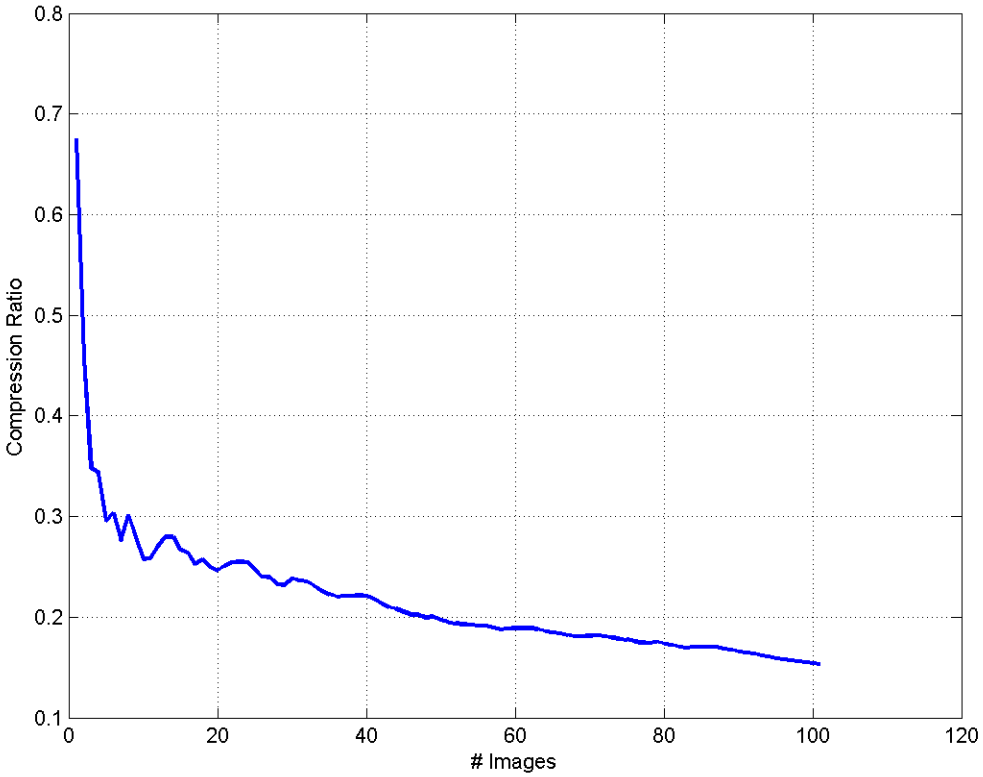

Figure 4.3: Compression Ratio, Scenario I

4.3.3 Scenario II

4.3.3.1 Parameters

For the LSH amplification parameters we set K equal to 30 and L equal to 3. We let α be distributed according to the DCT coefficient weights, and β uniform around the mean with range equal to $\frac{1}{2}$ standard deviation of the corresponding DCT coefficient. Again, we set the threshold θ to 2.

4.3.3.2 Running Time

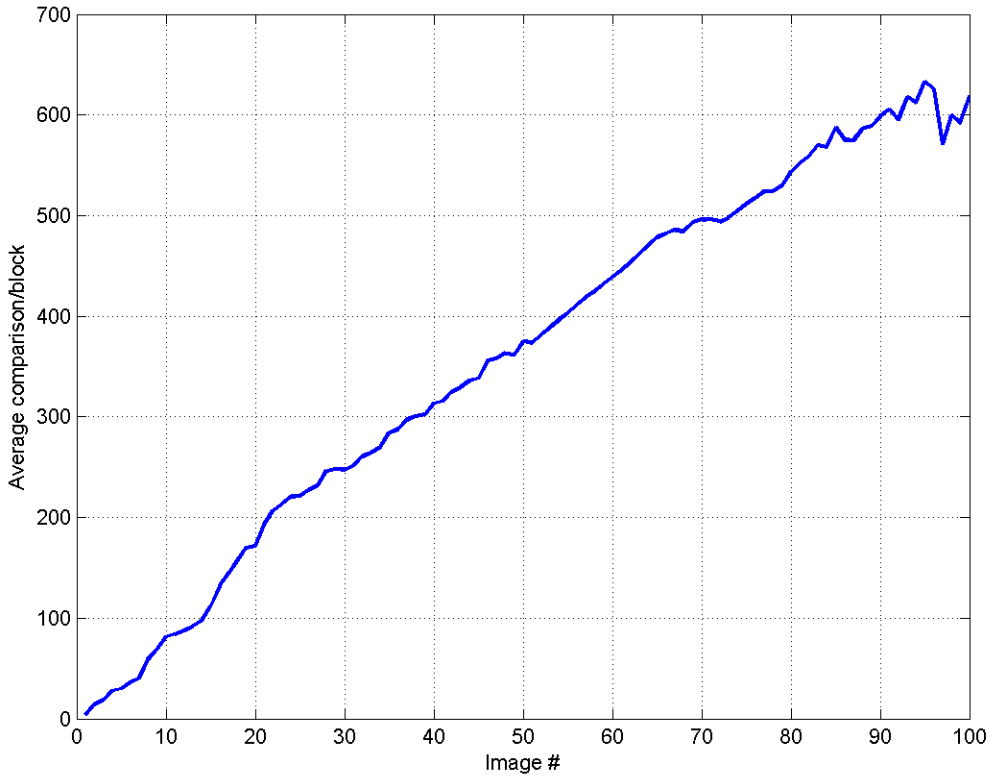

Figure 4.4: Average Comparisons per block, Scenario II

4.3.3.3 Compression

Figure 4.6 shows the compression ratio decreasing, reaching a final value of 0.1 in this case.

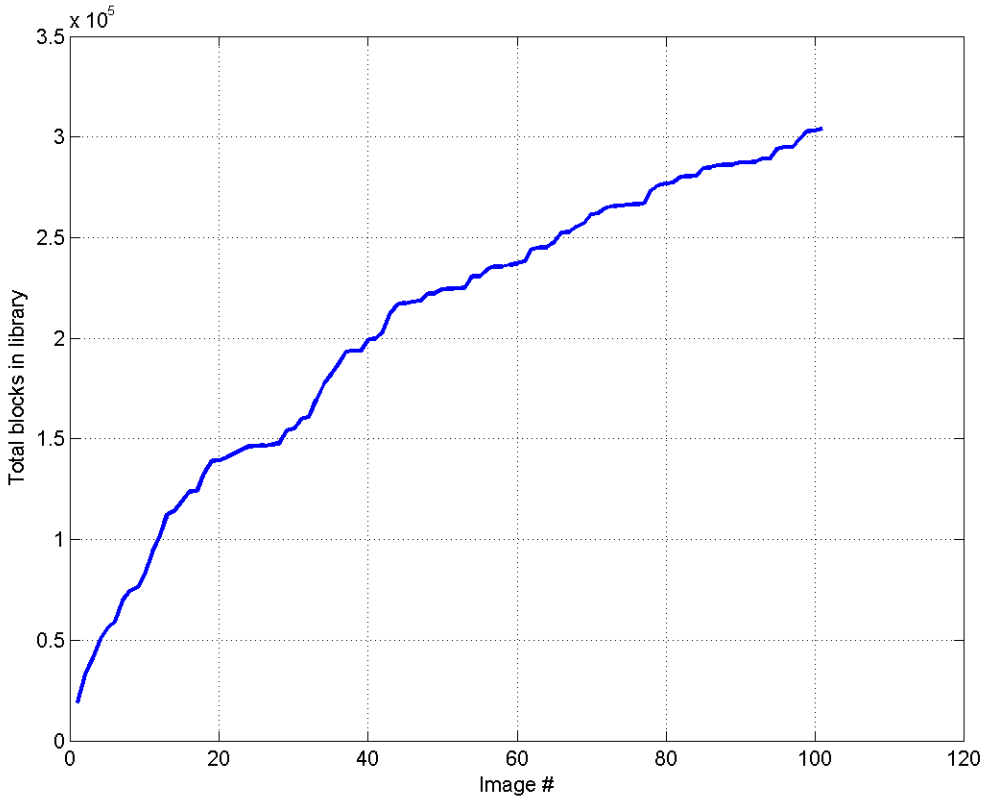

Figure 4.5: Number of blocks in library, Scenario II

4.3.4 Scenario I vs. Scenario II

4.3.4.1 Running Time

Choosing β uniform over the appropriate range provides better running time (Scenario II), within a constant factor. It can be seen in figure 4.7 that both curves grow similarly. It is also interesting to see how much the LSH structure reduced our search space: If we had to compare with every block in the library at each step, the running time would be much worse, and this is clearly visible in figure 4.8.

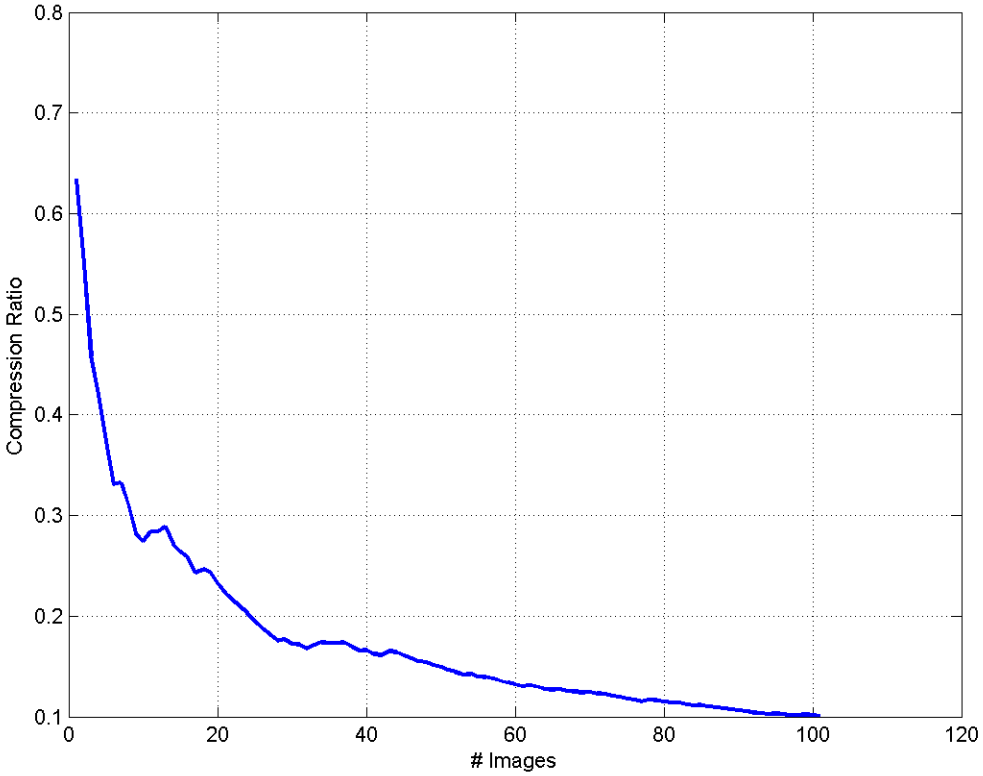

Figure 4.6: Compression Ratio, Scenario II

4.3.4.2 Compression

Comparing both scenarios for compression in figure 4.9, it is clear that picking β uniform over the appropriate range provides better performance. This is no surprise since this choice of β ensures that our buckets maintain the 'distance structure' as previously mentioned: blocks that are close will hash to the same bucket with high probability. Comparing the compression ratio of both scenarios, the first one provides $C = 0.16$, whereas the second one $C = 0.1$.

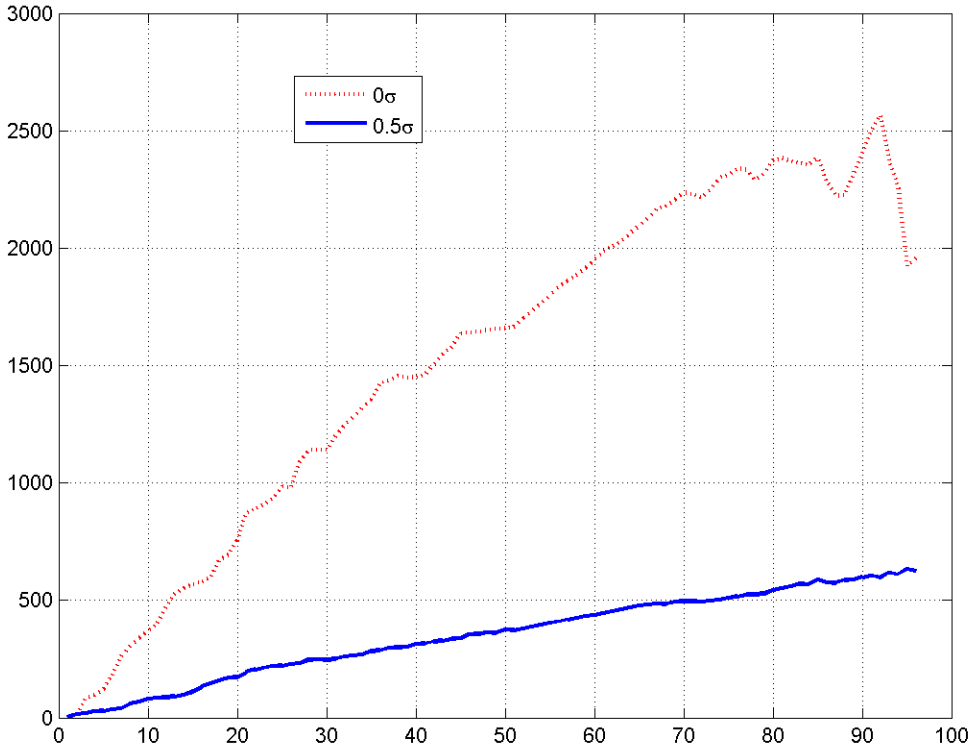

Figure 4.7: Average comparison per block : Scenario I vs. Scenario II

4.3.4.3 Visual Quality

We can see some of the visual artifacts of our algorithm, with a threshold $\theta = 2$, such as blocking in the regions where redundancy is most present. Scenario II in figure 4.12 provides slightly better visual quality than scenario I in figure 4.11, since we have higher probability of finding a closer block, given the structure of buckets it entails.

We conclude that applied on this artificial database, our algorithm compresses very

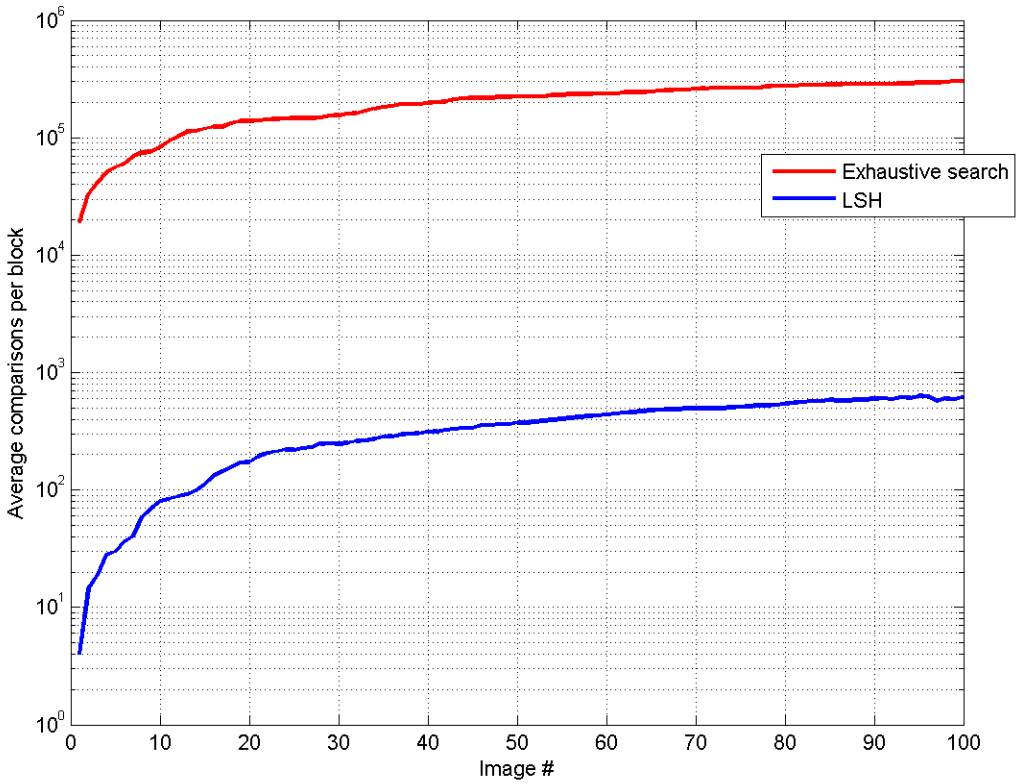

Figure 4.8: Average comparison per block : LSH vs. Exhaustive

well, as expected. We were able to reduce the size up to 10 times the original size of the uncompressed images. Another observation about β can be made: Making β uniform over a desired range helps us in finding additional blocks of interest, which in turn allows us to better compress the database given our threshold.

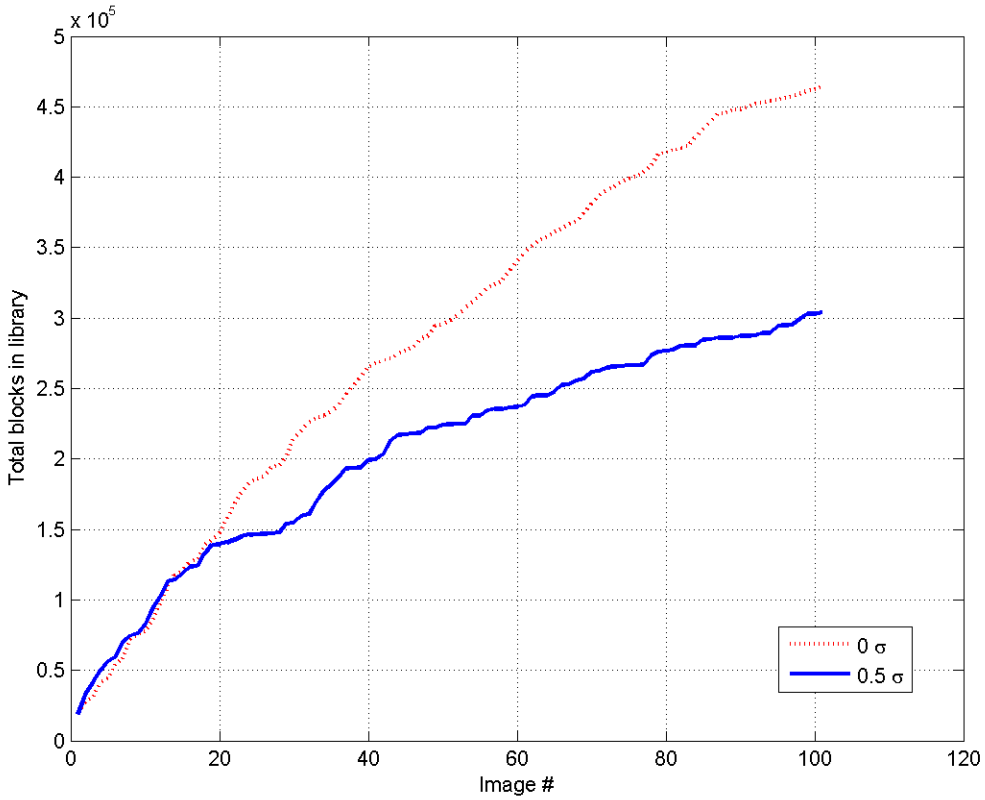

Figure 4.9: Blocks in library: Scenario I vs. Scenario II

4.4 Oxford database results

In this section, we consider a database of $1,502$ images, taken at the University of Oxford. The pictures do not look visually similar except in small regions such as sky or grass. For this dataset, we do not expect the same gains achieved in section 4.3, since pictures share much less redundancy in this case.

Figure 4.10: Original Uncompressed image

4.4.1 Experiment with variable β distributions

4.4.1.1 Parameters

We will consider three different scenarios, each sharing the same $K = 30, L = 3, \theta = 2$, and α parameters, but different β distributions. We let β be uniform with three different ranges: $0\sigma, 0.5\sigma$, and 1.5σ, where σ is the standard deviation of the corresponding DCT coefficient.

4.4.1.2 Running Time

Simulations highlight a relation between the running time and distribution of β, as one would expect. Intuitively, this is due to the bucket sizes that different choices of β entail. Given the very particular distributions of DCT coefficients, one

Figure 4.11: Compressed image using scenario I parameters

can see that increasing the range of β over some value would cause the bucket sizes to increase, as we are less likely to partition all the blocks into uniform buckets. This observation is verified by simulating our algorithm with different choices of β distributions, as shown in figure 4.13. For the 1.5σ case, the average number of comparisons per block is significantly larger than the other cases with smaller range.

4.4.1.3 Compression

As already mentioned, we expect the buckets to become more dense with 'close' points if we increase the range of β in a reasonable way. Although this might cause slower running time, it provides slightly better overall compression. Figure

Figure 4.12: Compressed image using scenario II parameters

4.14 shows the compression ratio of the three different scenarios: The largest range of β outperforms the two others, with final compression ratios of $0.66, 0.7,$ and 0.73 respectively. Note that since the images do not share a lot of redundancy, we initially experience an oscillating phase which rapidly reaches an average measure of redundancy inside the image, and then starts slowly decreasing due to the library becoming richer and our algorithm exploiting the redundancy it can find.

4.5 Extended Oxford database

In this section we run our algorithm on an extended dataset of section 4.4, where we include around $2,500$ extra photos. In this case, we compare two scenar-

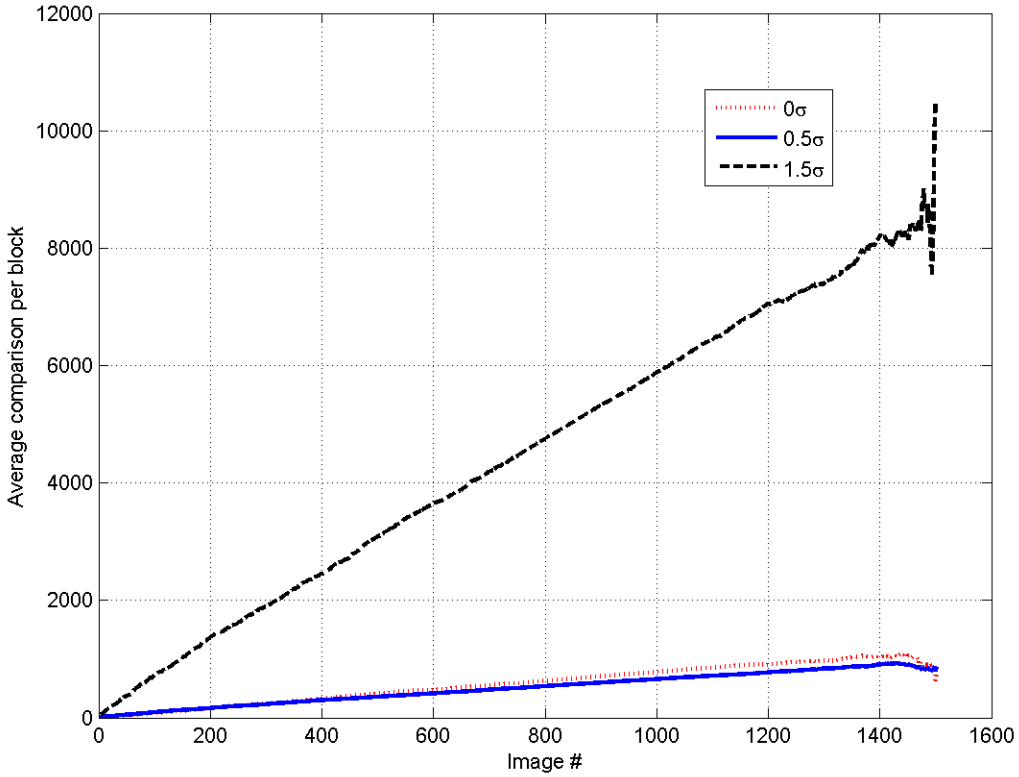

Figure 4.13: Average comparison per block : different β distributions

ios: one with distortion $\theta = 2$, and the other with $\theta = 0.5$. Here again, we let K equal to 30 and L equal to 3. Parameter α is distributed according to the DCT coefficient weights, and β uniform with range 0.5σ, where σ is the standard deviation of the corresponding DCT coefficient. Since $\theta = 0.5$ causes slower running time, we are satisfied with the statistics of the first $3,000$ images of our dataset.

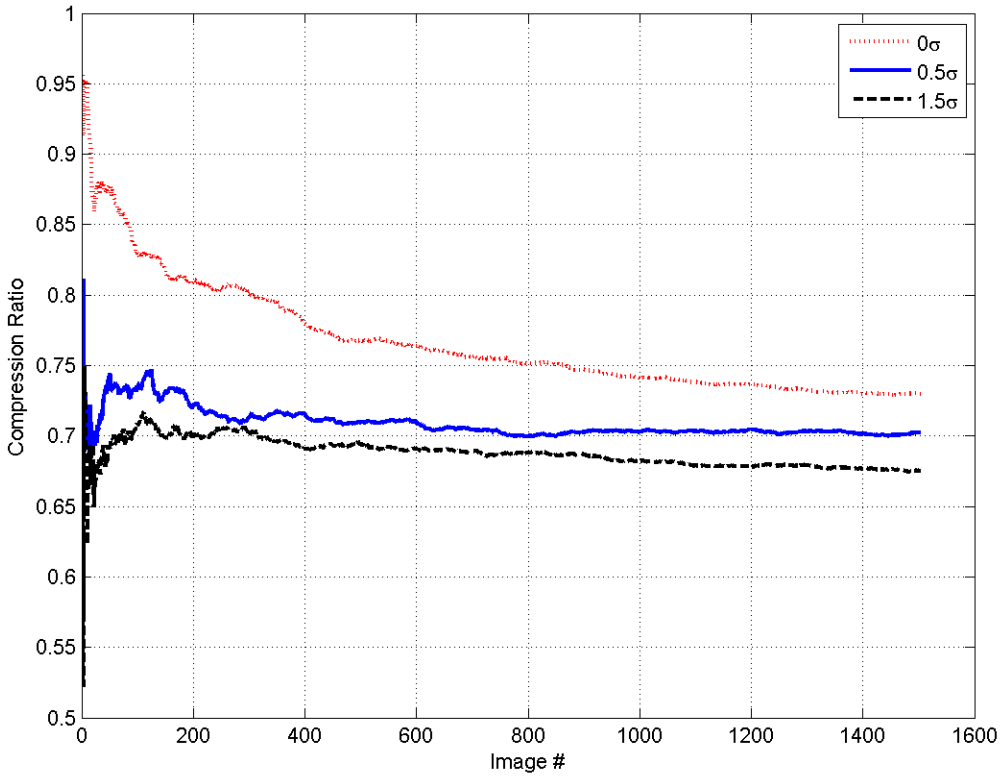

Figure 4.14: Compression Ratio: different β distributions

4.5.1 Running Time

With a lower threshold, the final library size is expected to be larger than with a higher threshold, as we are being more restrictive on the requirement of referencing a block by another. This said, we expect the running time to be much worse for $\theta = 0.5$. This can be observed in figure 4.15, where the lower threshold value requires much more comparisons per block, due to the larger library size.

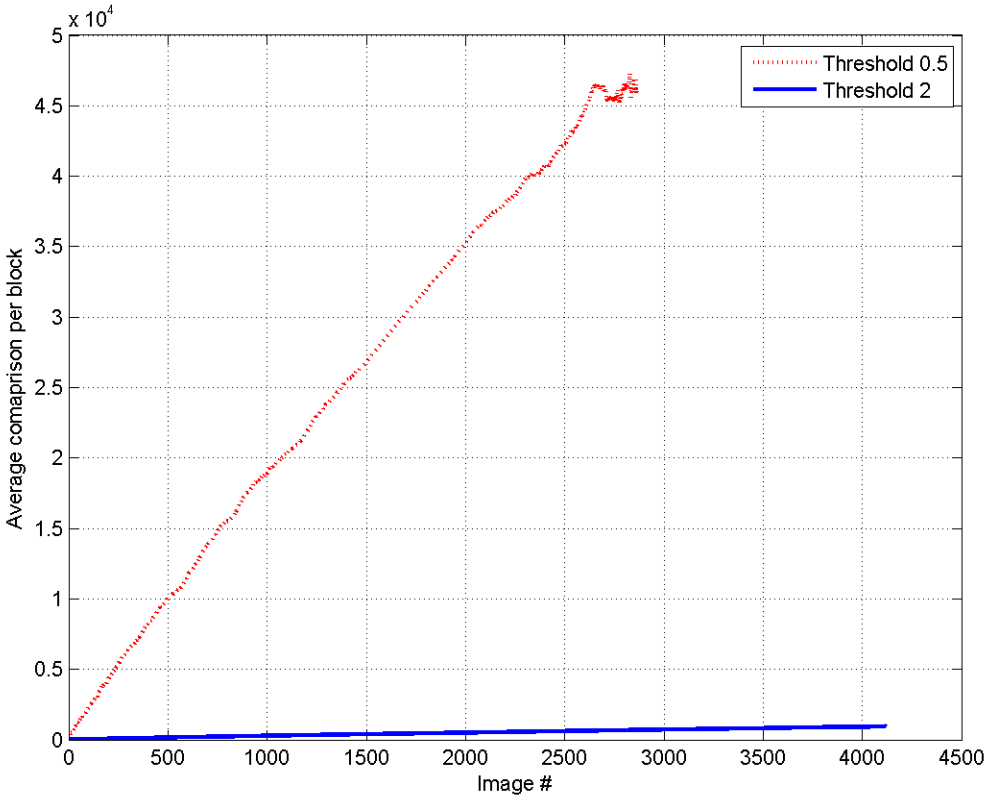

Figure 4.15: Average comparison per block : different θ values

4.5.2 Compression

Since the overall size of our library increases with a smaller threshold, the compression ratio is also believed to be higher *(worse)*. This is very natural since we get better image quality with lower threshold, thus we need to store more information. Figure 4.16 shows that for a threshold of 0.5, the final compression ratio is around 0.86, whereas for a threshold of 2, we achieve a 0.66 compression ratio.

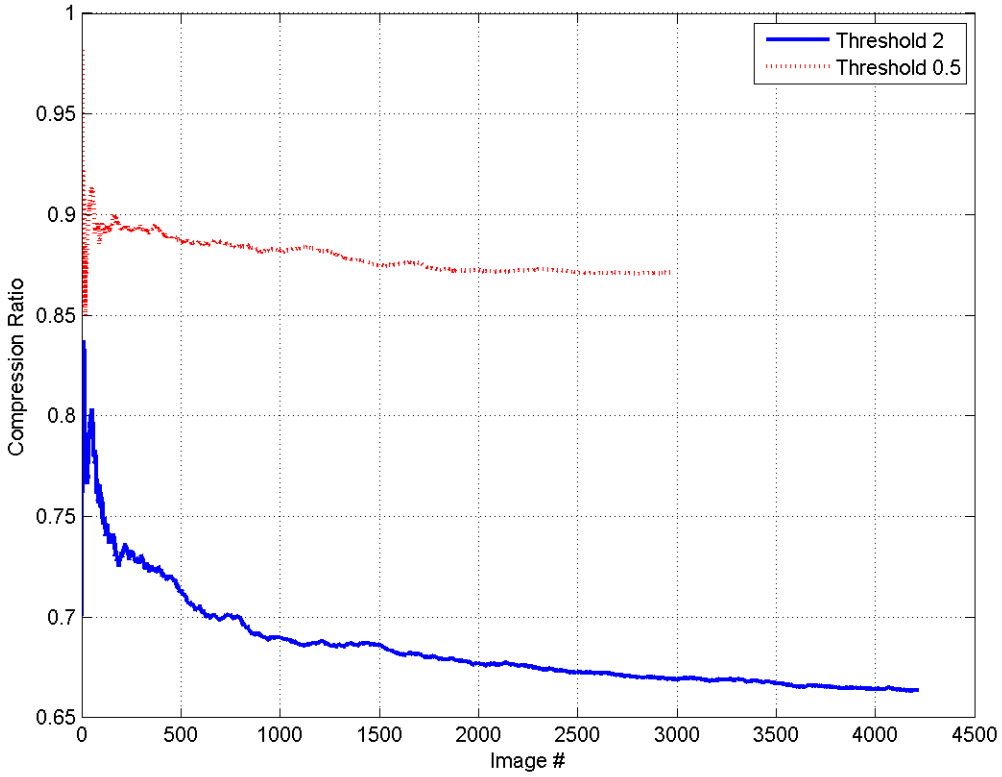

Figure 4.16: Compression Ratio : different θ values

4.5.2.1 Visual Quality

Although the lower threshold resulted in worse compression ratio and running time, we expect the images to look much more visually appealing which we find to be the case. The blocking effect is almost non-existent in figure 4.17, especially compared to the artifacts experienced in figure 4.18.

Figure 4.17: Image with $\theta = 0.5$

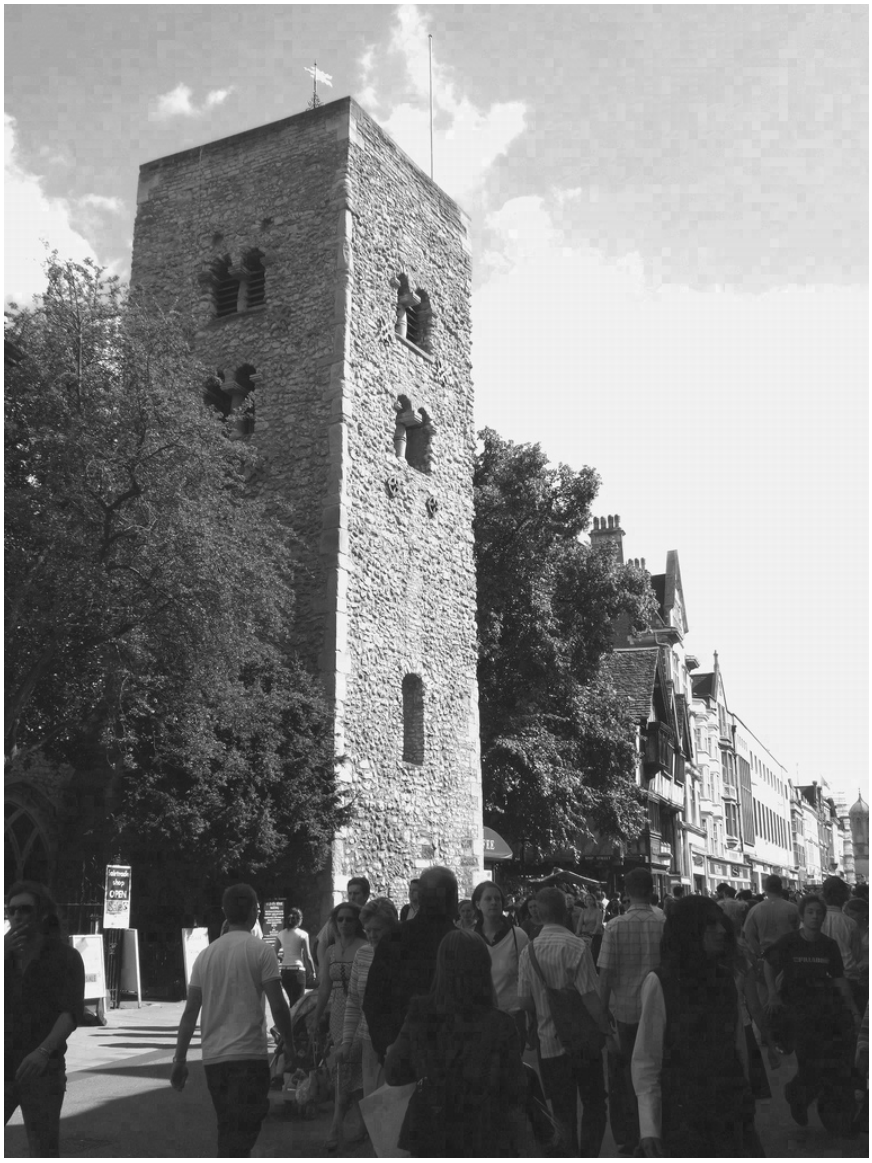

Figure 4.18: Image with $\theta = 2$

Chapter 5

Conclusion

In conclusion, we designed and implemented an efficient image database compression tool on C++. We showed that locally sensitive hashing can be used to efficiently search for similar blocks, a major component of our compression algorithm. Our system outputs a library of blocks Q and a list of pointers P as files, which can be used to form the images that were originally compressed. We showed that our algorithm works very well for databases of very similar images, where we were able to compress up to 90% of the original size. We simulated our system by varying different design parameters, and highlighted the trade-off that one might face between running time and compression size and quality.

Since there are a lot of components and design alternatives that one can consider, there is still plenty of room for potential improvement. One particular problem we face in some scenarios is the 'blocking' effect after compressing with a certain threshold. One could possibly suggest a different distance measure other than the L_1 norm that could account better for visual similarity. Another extension to this project could be parallelizing this tool over different machines and managing memory/disk access for very large databases.

Bibliography

[1] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975. [Online]. Available: http://doi.acm.org/10.1145/361002.361007

[2] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proceedings of the 25th International Conference on Very Large Data Bases*, ser. VLDB '99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 518–529. [Online]. Available: http://dl.acm.org/citation.cfm?id=645925.671516

[3] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, ser. STOC '98. New York, NY, USA: ACM, 1998, pp. 604–613. [Online]. Available: http://doi.acm.org/10.1145/276698.276876

[4] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, ser. STOC '02. New York, NY, USA: ACM, 2002, pp. 380–388. [Online]. Available: http://doi.acm.org/10.1145/509907.509965